**BST Project: Code Documentation**

Brandon Michelsen

**Introduction:**

Binary Search Trees – or BSTs – are a very useful data structures when there are many elements to search through. As you will soon see, the structure of BSTs allows for very fast searching, at least in most cases. A programmer can use a BST for data storage in situations such as local databases since there is such a good search time. BSTs can also serve as a starting point to learn more about graph data structures, as there are elements that are similar amongst both forms of data structures.

Much like linked lists, data elements of the BST are stored in nodes. However, instead of the nodes having pointers to the node before the current node and the node after, nodes in a BST have pointers to two child nodes: a left child node and a right child node. These child nodes are what form the structure of a BST and are also what makes searching so fast. Each child node develops into its own subtree from the node above it. A BST also contains a root node, which is where the whole tree grows from. Following the subtrees, it can be seen that the nodes to the left contain data elements that are smaller than those above them, while nodes to the right contain elements that are larger. In other words, the left subtree will contain nodes that are lesser in value than the current node, while the right subtree contains nodes that are greater. This is what makes search times so fast for BSTs. Every time a new node is encountered in the search algorithm, it cuts the effective number of elements to search through in half.

The best way to show this is to look at an example. Assume there is a tree containing the following elements – 10, 5, 4, 3, 6, 16, 17, 18, 15 – with 10 being the root node. The value to be searched for is 6. The algorithm starts with the root node, checking to see if the value of that node is greater than, less than, or equal to the value to be found, which is 6. If the value is less than 6, the algorithm will search to the right subtree and if it is greater it will search to the left subtree. Since 10 is greater than 6, the algorithm will search to the left, cutting out the whole right subtree. The next node will be 5. In this case, the algorithm will veer right, cutting out the left subtree of the node containing 5. The next node contains 6, meaning the algorithm has found the proper node. As can be seen, every time a new node is searched, a subtree is cut out of the algorithm, effectively shrinking the number of nodes to traverse with each node checked. This is what makes a binary search tree so fast.

There are three main operations a BST must perform: insertion, deletion, and searching. The searching algorithm operates much as described above, checking each encountered node and choosing the correct subtree to traverse based on that check. The insertion operation operates in a very similar manner to the searching operation, except that it looks for an empty node at which to insert a node instead of finding an already existing node. The insertion operation also ensures that no duplicate values are stored in the tree, as this could break the structure of the tree.

Deletion is a bit more difficult than the first two operations, as there are multiple cases that it must check for. These cases are as follows: deletion of a node with no children (a leaf node), deletion of a node with only one child, and deletion of a node with two children. Deletion of a node with no children is by far the easiest case. First, the node to be deleted needs to be found. Then, that node is removed from the tree and deleted. That is all there is to it. The next case is a bit more difficult to handle. In this instance, the parent node is involved. First, it needs to be determined if the node to be deleted is the left child or the right child of its parent node. Once this is determined, the child of the node to be deleted will be set as the new child node of the parent node. If the node to be deleted was the right child node, the parent’s right pointer will point to the child node of the node to be deleted. Otherwise, the parent’s left pointer will point to the child node. Then, the node to be deleted can be removed from the tree and deleted.

The final case is the most difficult to manage. In this case, we need to find a new node to replace the node to be deleted with that will not break the tree. There are two widely used methods to find a replacement node. The first method is to look to the left subtree and find the node with the greatest value. The second is to look to the right subtree and find the node with the smallest value. In either case, the node to be deleted will be replaced with a value that fits that spot in the tree. Now, what is meant by that is that the replacement node must be greater that all the values in the left subtree and less than all the values in the right subtree. The two methods mentioned find nodes that fit this requirement. After a proper replacement node is found and added into the proper spot in the tree, the node to be deleted can be removed and deleted.

As a final note, though a BST can make for fast search times, a particular BST may not always run searching algorithms very fast. For example, a BST can become skewed to one side or the other, which increases the search time of the entire tree. To avoid this, a Self-balancing BST can be used. This type of tree will restructure the tree with every insertion and deletion, maintaining balance on both sides of the root. This makes it possible to keep the fast search times for the entire tree. However, there is a performance degradation with insertion and deletion. Because of this, it is up to the programmer to determine which tree structure is the best for the situation.

**Runtime Analysis:**

Typically, Binary Search Trees have a search algorithm whose best case runtime is O(log N). The worst-case scenario for search would be O(N), as every element in the tree will need to be searched (this is true for highly skewed trees, such as trees that only have all left nodes). As for average case, that would depend on the size – or height – of the tree, O(tree height). Actually, all of the runtime cases can be brought down to an analysis of the height of the tree. For example, in the worst-case scenario, the height of the tree is the same as the number of elements, N. This is why it has a time complexity of O(N). In a similar case, the best-case time complexity has a logarithmic relationship to the number of elements, giving it a time complexity of O(log N).

For my analysis, I have a chart below showing the calculations of the logarithmic value for different values of N:



Here is the amount of time it took for each of those values for N:



These tests were run using a populated vector of unsorted data. As can be seen, the runtime for these test cases is fairly quick, remaining under a second.

These tests show that the runtime does not have a linear relationship to the number of elements, meaning the runtime does not directly increase due to an increase in the number of elements. Instead, the runtime shows a more logarithmic relationship, only increasing a small amount with the number of elements.

**Programmer’s Guide**

This section gives information on how the classes and structs used in this program are meant to be used.

**Tree Node Struct:**

|  |
| --- |
| **Struct: TreeNode** |
| template<class T> |
| **Member Variables:** |
| T data  TreeNode<T>\* left  TreeNode<T>\* right |

The TreeNode struct is a data structure that stores data elements of type “T” (generic data type) along with pointers to left and right child nodes. There is direct access to the member variables.

**Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| data | Data to be stored in the node (of type “T”) |
| left | Pointer to the left child node (of type TreeNode<T>\*) |
| right | Pointer to the right child node (of type TreeNode<T>\*) |

**Binary Search Tree Class:**

|  |
| --- |
| **Class: BinarySearchTree** |
| template<class T> |
| **Member Variables:** |
| TreeNode<T>\* root |
| **Member Functions (Public):** |
| BinarySearchTree()  void setRoot(TreeNode<T>\* rootPtr)  TreeNode<T>\* getRoot()  void insertNode(T inData)  void deleteNode(T delData)  TreeNode<T>\* max()  TreeNode<T>\* min()  TreeNode<T>\* find(T dataToFind) |
| **Member Functions (Private):** |
| void insertNodePrivate(TreeNode<T>\*& node, T inData)  void deleteNodePrivate(TreeNode<T>\*& node, T delData)  void removeRoot();  void removeMatchingNode(TreeNode<T>\*& parent, TreeNode<T>\* current)  TreeNode<T>\* findMin(TreeNode<T>\* start) |

The BinarySearchTree class is a tree data structure that stores data elements of type “T” (generic data type). Each data element is stored in a node of type TreeNode<T> (from the TreeNode struct). The root of the tree is stored as a pointer as part of the BinarySearchTree class. Insertion of nodes will always start by looking at the root node, and then traversing the tree until a proper insertion point is found (an empty node that is in the proper subtree). Deletion will first search for the value in the tree and will only then remove the proper node.

The public insertion and deletion functions have helper functions that are private to the class. These functions serve to help lessen the code complexity of the functions.

**Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| root | Points to the root node (the very first node) in the tree (of type TreeNode<T>\*) |

**Public Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| (constructor) | Constructs the Binary Search Tree object | None | None |
| setRoot | Sets the value of the root member variable | TreeNode<T>\* rootPtr | void |
| getRoot | Retrieves the value of the root member variable | None | TreeNode<T>\* |
| insertNode | Inserts a new node into the tree | T inData | void |
| deleteNode | Deletes a node from the tree | T delData | void |
| max | Returns the node with the maximum value in the list | None | TreeNode<T>\* |
| min | Returns the node with the minimum value in the list | None | TreeNode<T>\* |
| find | Searches for and returns the node containing the value being searched for | T dataToFind | TreeNode<T>\* |

**Private Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| insertNodePrivate | Helper function for inserting nodes | TreeNode<T>\*& node  T inData | void |
| deleteNodePrivate | Helper function for deleting nodes | TreeNode<T>\*& node  T delData | void |
| removeRoot | Properly removes the root from the tree | None | void |
| removeMatchingNode | Properly removes a node from the tree | TreeNode<T>\*& parent  TreeNode<T>\*& current | void |
| findMin | Finds the minimum value in a given subtree | TreeNode<T>\* start | TreeNode<T>\* |

**Implementation Details:**

The find, min, and max functions return pointers the nodes that they find. Because of this, it is up to the programmer to check for null pointers and properly deal with them.

**Usage Examples:**

Below are the usage examples for the public member functions of the class. For each example, ensure that you are using the proper directory where the BST.h file is located.

**Root Getter and Setter**



**Insertion Function**



**Deletion Function**



**Find Function**



**Max Function**



**Min Function**

